

Safe and Static Methods for Memory Management

John C F

Supervisor:

Dr. Amitabha Sanyal

Dept. of Computer Science & Engg.
Indian Institute of Technology, Bombay

May 2016

Contents

1	Introduction	2
2	Region Based Memory Management	3
3	Region Inference	5
4	User-defined Regions	9
5	Linear Type System	12
	Conclusion	19
	References	20

Chapter 1

Introduction

Historically, all memory management was done manually, where the programmer would be supposed to carefully allocate and free memory as needed. This could result in many bugs due to the human-nature of programmers (which is to err!) such as:

1. Accessing unallocated memory regions (use-after-free, out-of-bounds)
2. Forgetting to free unused memory regions (memory leaks)

Use-after-free behavior usually causes stability issues (the more severe, the better!), but the subtle ones might open up terrible security holes. Minimizing memory leaks is especially crucial for long running programs. In embedded systems, memory might be very limited, and to make matters worse, the operating system may not have process isolation features (virtual memory).

Therefore, due to obvious reasons, automatic memory management techniques were researched. The most commonly used techniques involve a runtime routine which actively scans for allocated but “stale” memory regions (garbage) and frees them. These techniques are collectively termed Garbage Collection, and it usually involves a per-“object” reference counting along with reachability tracing. Tracing procedure is compute-intensive, as well as it requires that the references don’t change while it does the trace (stop-the-world). Although the tracing routines is infrequently invoked at runtime (when memory usage crosses a certain threshold), it would still incur unacceptable levels of latencies for real-time applications. So the question is: can we do less at runtime and still be safe from memory bugs?

Two major approaches were formulated to address this question:

- *Regions*: An extended stack-based discipline to support dynamic sized types.
- *Linear types*: A type system that ensures that objects are used (or consumed) exactly once. Thus simplifying resource management.

In chapters that follow, many example programs and associated errors are discussed. All errors under discussion are caught during static analysis.

Chapter 2

Region Based Memory Management

In this model (as described by Tofte and Talpin[1]), the memory is thought of as a stack of regions. And each region can grow indefinitely as values are put into it. All regions are associated with some lexical scope (a code segment) which is responsible for deallocating the whole region when the program exits that scope. The scheme we will discuss involves translation of source language into a region annotated target language using a technique called region inference. The region inference methodology discussed is provably memory safe; that is, values are associated with regions in such a way that there are no dangling pointer dereferences.

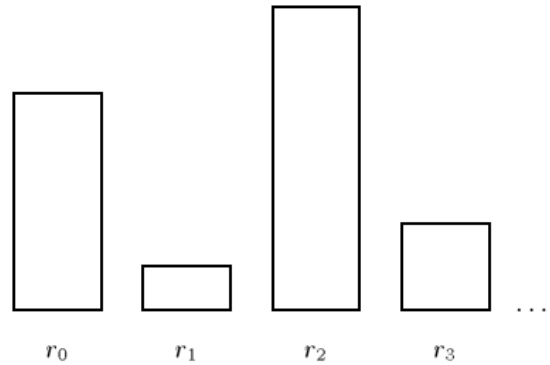


Figure 2.1: Stack of regions

It is important to note that regions can only grow and do not shrink – that is, values can only be put into it, and cannot be individually freed from it. Also, it should be emphasized that, in this scheme, the programmer cannot manage regions, and has no control over which values are put into which regions. Region Inference takes care of all association of values with regions. Before diving into the details of region inference, here's a small example of the source language and the target language:

```
1 | let x = (2, 3) in ( $\lambda y$ .(fst x, y)) end 5
```

which translates to:

```
1 | letregion  $\rho_4$ ,  $\rho_5$ 
2 | in letregion  $\rho_6$ 
3 |   in let x = (2 at  $\rho_2$ , 3 at  $\rho_6$ ) at  $\rho_4$ 
4 |     in ( $\lambda y$ .(fst x, y) at  $\rho_1$ ) at  $\rho_5$ 
5 |     end
6 |   end
7 |   5 at  $\rho_3$ 
8 | end
```

Points of note

- All values are boxed (live behind a reference). That is, all constants and variables in the program are treated as a reference to the some location (in the heap) managed by regions.
- `fst` takes (a reference to) a pair and returns (a reference to) its first element.
- `e at  $\rho$` means “store the value produced by `e` in the region ρ ”.
- The only way of introducing and destroying regions is using `letregion` blocks.
- Region variables ρ_1 , ρ_2 and ρ_3 , occurring free in the above code, will contain the final result.
- The closure (lambda expression) itself is stored at ρ_5 , which when applied on an argument (binding y to a reference) will store a pair at ρ_1 and return (a reference to) that pair.
- x escapes the `let` block with the closure (i.e. the closure holds a reference to x).
- Part of x (`3 at  $\rho_6$`) is deallocated and a dangling pointer is formed (see figure 2.2), but it is safe since it will not be dereferenced.
- Region variables ρ_4 and ρ_5 may be bound to the same region.

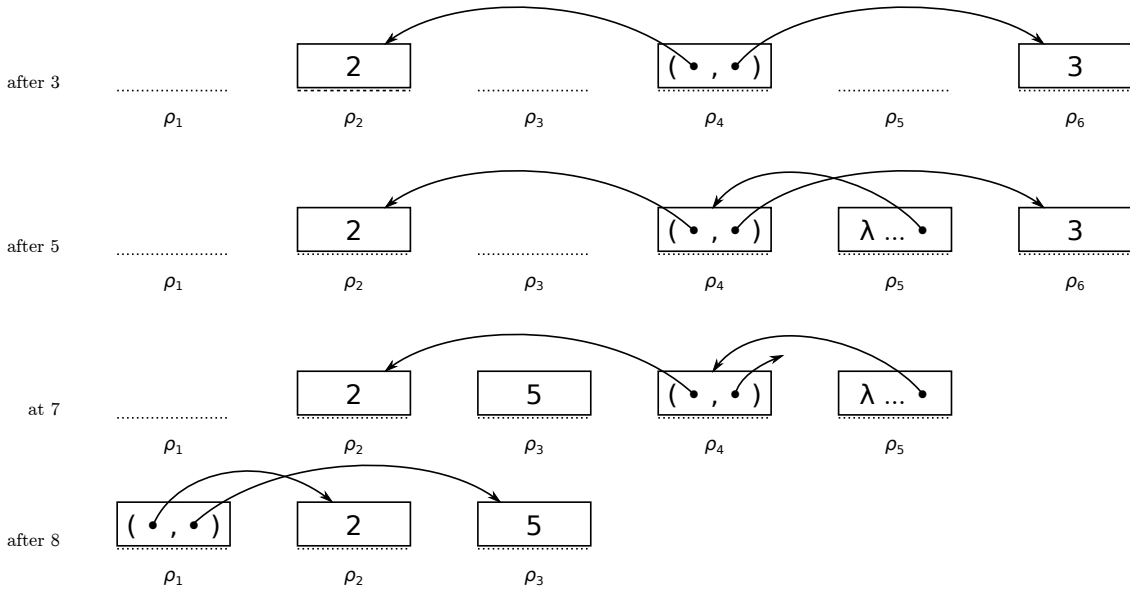


Figure 2.2: State of regions at various stages of execution

Chapter 3

Region Inference

In this chapter, we'll take a detailed look at the translation using the example from before:

```
1 | let x = (2, 3) in (λy.(fst x, y)) end 5
```

The most interesting part in the translation was that it was inferred that x escaped the **let** block as well as that part of x could be thrown away even before the closure was applied! Before we dive into the algorithm, let me give you a rough idea of the type system of the target language.

Below, we will be discussing the type, “place”, and “effect” of various expressions in the target language. “ $e :: \tau, \rho$ w.e. ϵ ” is read “expression e produces a value of type τ at place ρ with effect ϵ ”. A place simply means a region; within the scope of our discussion, “place” and “region” may be used interchangeably. Effect indicates the set of regions accessed when the expression e is executed. The effect may be empty, or may contain one or more of $\text{get}(\rho)$'s or $\text{put}(\rho)$'s, which respectively indicate reading a value (involving a dereference) from or writing a new value (involving an allocation) to region ρ . We'll use a shorthand of $\{\text{get}(\rho_1, \rho_2)\}$ to indicate $\{\text{get}(\rho_1), \text{get}(\rho_2)\}$.

Let's take a look at how the type scheme (a.k.a polytype) of the built-in **fst**, which takes in a pair and returns the first element, might be represented in the target language:

$$\forall \rho_0 \rho_1 \rho_2 \forall \alpha_1 \alpha_2. ((\alpha_1, \rho_1) * (\alpha_2, \rho_2), \rho_0) \xrightarrow{\epsilon.\{\text{get}(\rho_0)\}} (\alpha_1, \rho_1), \rho'$$

$\forall \rho_0 \rho_1 \rho_2$ denotes that the function is region-polymorphic over three region variables, and $\forall \alpha_1 \alpha_2$ denotes that it is type-polymorphic too. $(\mu_1 * \mu_2, \rho)$ denotes the type of a pair stored at region ρ , where μ_1 denotes the type and place of the first element, and μ_2 denotes that of the second. $\epsilon.\{\text{get}(\rho_0)\}$ above the arrow denotes that, when applying the function, a value from region ρ_0 will be read. And finally, ρ' is the place at which this function is stored. For the sake of simplicity, we'll assume that **fst** is monomorphic to match the context in which it appears, and also ignore the effect of $\{\text{get}(\rho')\}$ when **fst** is applied.

The Algorithm

The Region Inference algorithm roughly involves the following steps:

1. Annotate all value-producing expressions with distinct region variables:

```

1 | let x = (2 at  $\rho_1$ , 3 at  $\rho_2$ ) at  $\rho_3$ 
2 | in ( $\lambda y.(\text{fst } x, y)$  at  $\rho_4$ ) at  $\rho_5$ 
3 | end
4 | 5 at  $\rho_6$ 

```

2. Find the type and effect of all expressions (in a bottom-up fashion):

$$\text{fst} :: ((\text{int}, \rho_1) * (\text{int}, \rho_2), \rho_3) \xrightarrow{\epsilon.\{\text{get}(\rho_3)\}} (\text{int}, \rho_1), \rho' \\ \text{w.e } \emptyset$$

The type of **fst** shown above is the type of **fst** in that context. “w.e $\emptyset$ ” indicates that the expression “**fst**”, all by itself, does not access any regions. A function reference before being applied to actual arguments does not have any effect.

$$(2 \text{ at } \rho_1, 3 \text{ at } \rho_2) \text{ at } \rho_3 :: (\text{int}, \rho_1) * (\text{int}, \rho_2), \rho_3 \\ \text{w.e } \{\text{put}(\rho_1, \rho_2, \rho_3)\} \\ \text{fst } x :: \text{int}, \rho_1 \\ \text{w.e } \{\text{get}(\rho_3)\}$$

Note here that the effect of this application is what appeared above the arrow in the type definition of **fst**. A subtle point to notice is that there is no **get**(ρ_1) effect. This is because **fst** only needs to read the value of the pair definition to get the reference to the first value (which is returned without being dereferenced).

$$(\text{fst } x, y) \text{ at } \rho_4 :: (\text{int}, \rho_1) * (\text{int}, \rho), \rho_4 \\ \text{w.e } \{\text{get}(\rho_3), \text{put}(\rho_4)\}$$

The most interesting step is up next. The effect of the expression above goes into the type of the closure itself (above the arrow along with an effect variable ϵ). This is the key idea in escape analysis. Even though ρ_3 is neither part of the arguments, nor part of the return type, we’ll still know that it *will be* accessed by the closure. The effect variable ϵ is used during monomorphisation of region- (and effect-) polymorphic functions. It is not relevant in this example, and we won’t be going into any details of it.

$$(\lambda y.(\text{fst } x, y) \text{ at } \rho_4) \text{ at } \rho_5 :: (\text{int}, \rho_6) \xrightarrow{\epsilon.\{\text{get}(\rho_3), \text{put}(\rho_4)\}} ((\text{int}, \rho_1) * (\text{int}, \rho_6), \rho_4), \rho_5 \\ \text{w.e } \{\text{put}(\rho_5)\}$$

Note that since lambda expressions are non-recursive and can only be applied once, the type and place of parameters can be directly inferred from the type and place of actual arguments. This is why y has type (int, ρ_6) .

$$\text{let } x = \dots \text{ in } \dots \text{ end} :: (\text{int}, \rho_6) \xrightarrow{\epsilon.\{\text{get}(\rho_3), \text{put}(\rho_4)\}} ((\text{int}, \rho_1) * (\text{int}, \rho_6), \rho_4), \rho_5 \\ \text{w.e } \{\text{put}(\rho_1, \rho_2, \rho_3, \rho_5)\}$$

3. If a region variable occurs free in an expression but does not occur in its type or place (it may occur in the effect), then that expression can be surrounded by a **letregion** block with that associated region.

Notice in the above expression that ρ_2 does not occur in the type and place. Therefore the above block can be wrapped in a **letregion** involving ρ_2 .

```

1 | letregion  $\rho_2$ 
2 | in let x = (2 at  $\rho_1$ , 3 at  $\rho_2$ ) at  $\rho_3$ 
3 |   in ( $\lambda y$ .fst x, y) at  $\rho_4$ ) at  $\rho_5$ 
4 |   end
5 | end

```

$$\begin{aligned}
 (\text{above expression}) :: (int, \rho_6) &\xrightarrow{\epsilon.\{get(\rho_3), put(\rho_4)\}} ((int, \rho_1) * (int, \rho_6), \rho_4), \rho_5 \\
 &\text{w.e } \{\text{put}(\rho_1, \rho_3, \rho_5)\}
 \end{aligned}$$

Note that ρ_2 is removed from the effect set. This is because, ρ_2 is introduced and destroyed within the expression, and the rest of the program need not be aware of it.

4. Repeat steps 2 and 3 for larger scopes:

$$\begin{aligned}
 \text{letregion } \rho_2 \text{ in ... end 5 at } \rho_6 &:: (int, \rho_1) * (int, \rho_6), \rho_4 \\
 &\text{w.e } \{\text{get}(\rho_3), \text{put}(\rho_1, \rho_3, \rho_4, \rho_5, \rho_6)\}
 \end{aligned}$$

Notice here that ρ_3 and ρ_5 does not appear in type and place, and therefore it does not escape out of this expression. And thus by wrapping the whole expression in a **letregion** introducing those region variables, we'll get the aforementioned final target code.

```

1 | letregion  $\rho_3, \rho_5$ 
2 | in letregion  $\rho_2$ 
3 |   in let x = (2 at  $\rho_1$ , 3 at  $\rho_2$ ) at  $\rho_3$ 
4 |     in ( $\lambda y$ .fst x, y) at  $\rho_4$ ) at  $\rho_5$ 
5 |     end
6 |   end
7 |   5 at  $\rho_6$ 
8 | end

```

$$\begin{aligned}
 (\text{above expression}) &:: (int, \rho_1) * (int, \rho_6), \rho_4 \\
 &\text{w.e } \{\text{put}(\rho_1, \rho_4, \rho_6)\}
 \end{aligned}$$

Limitations of automatic region inference

The major limitation of RBMM is that certain values may live longer than needed because it was put into long-living regions. For example, consider this hypothetical case:

```

1 | let x = (read_ints big_file)
2 | in let y = len x,
3 |     z = avg x
4 |     in long_running y z
5 |     end
6 | end

```


Which would be roughly translated to (assume that `read_ints[ $\rho_1$ ]` will parse a file and return a list of integers allocated at ρ_1):

```

1 | letregion  $\rho_1$ 
2 | in let  $x = (\text{read\_ints}[\rho_1] \text{ big\_file})$ 
3 |     in let ...
4 |         in long_running  $y \ z$  end
5 |     end
6 | end

```

This would mean that x will remain in memory while `long_running` is executed even though it cannot be referenced from it. A resource conscious programmer might want to explicitly free large data structures like those.

The above code could be rewritten in a “region-friendly” manner as:

```

1 | let ( $y, z$ ) =
2 |     let  $x = (\text{read\_ints} \text{ big\_file})$ 
3 |     in ( $\text{len } x, \text{avg } x$ )
4 |     end
5 | in long_running  $y \ z$ 
6 | end

```

Which will be translated to:

```

1 | let ( $y, z$ ) =
2 |     letregion  $\rho_1$ 
3 |     in let  $x = (\text{read\_ints}[\rho_1] \text{ big\_file})$ 
4 |         in ... end
5 |     end
6 | in long_running  $y \ z$ 
7 | end

```

This refactoring may not always be very straightforward. But this brings us to a major practical issue: the program structure has high influence on how well region inference performs [2]. Therefore, the programmer needs to know how region inference works even though s/he does not directly interact with regions. When the program gets complex, it can be hard to predict region inference.

One other issue with RBMM is concerning tail recursion. It is not always straightforward to do tail call optimization (for instance, it could be inferred that the return value must have the same region as one of the arguments). This can be mitigated by an ad hoc technique called Storage Mode Analysis (during code generation from target language), which resets a region during an allocation if it is found that the values in it are no longer “live”.

Despite the fact that region inference works really well for many programs, it is not always possible to have the optimal reclamation of dead memory. Therefore, in MLKit[3] (the first practical language with a complete region inference system), a tracing Garbage Collector (GC) was introduced [4] to minimize memory leaks. It was later proved that a certain type of tracing GC can be safely integrated with region inference [5].

Chapter 4

User-defined Regions

In MLKit, the user writes code agnostic of memory management method and the compiler automatically associates each value generating expression with a region of smallest possible lifetime subject to safety constraints.

In Cyclone [6], an imperative language which extends C dialect, the user introduces regions and associates values with regions manually, and the compiler simply try to prove safety of those associations or terminates with an error.

Cyclone was an attempt to achieve the following three seemingly conflicting goals:

- *Safe*: Provide memory safety, type safety, and thread safety
- *Static*: Safety should be guaranteed at compile-time
- *Explicit*: The programmer should easily be able to tell (or specify) when objects are allocated and deallocated (how they are managed)

By aiming to be Static and Explicit, an implicit goal of Cyclone is to not compromise on performance compared to C.

The main differences of Cyclone compared to MLKit were:

- *Manual region declaration and annotations*: This made memory management explicit.
- *Minimal region inference and default annotations*: Region inference was much simplified and limited; instead missing annotations were filled in using default annotation rules wherever possible. This helped reduce the number of annotations required, and easy to translate a piece of code written in C into Cyclone.
- *Region subtyping*: Since regions are lexical blocks, they cannot partially overlap. Therefore, within a function, regions would have outlives relations between each other. If ρ_1 outlives ρ_2 , then ρ_1 is a subtype of ρ_2 (ρ_1 is ρ_2 and more, just as a Cat is an Animal and more). Therefore a pointer `int * $\rho_1$` (an integer pointer to a value inside ρ_1) can be safely typecast to `int * $\rho_2$` . This is a key idea during region analysis to prove that a program is free of dangling pointer dereferences.
- *Simple effects*: There are no effect variables, and instead there is a type operator, `regions_of( $\tau$ )`, which gives the set of regions that occur in τ (a `struct` may have multiple pointers to values in various regions). A function, which takes arguments of types $\tau_1, \tau_2, \dots$, and returns τ_0 , is assumed to access regions $\bigcup_i \text{regions_of}(\tau_i)$ when executed.

Let us take a look at an example. Below, 'L: { ... } indicates introduction of a new region L, and new expr evaluates expr, stores it in the so-called heap region (which is assumed to live forever), and returns a pointer to it. rnew_L expr is similar, but it stores it in the region L. ρ_L is the region variable associated with the region L, and ρ_H is associated with the heap region.

```

1 void cp(int** pp1, int** pp2) {
2     *pp1 = *pp2;    // error
3 }
4
5 void foo(int c) {
6     int* x = new 5;
7     'L: {
8         int* y = rnewL 8;
9         if (c == 1)
10             cp(&x, &y);    // unsafe!
11     }
12     //read *x
13 }
```

The default annotations try to be as general as possible. Thus the fully verbose version of the above code is:

```

1 void cp< $\rho_1$ ,  $\rho_2$ ,  $\rho_3$ ,  $\rho_4$ >(int* $\rho_1$  *pp1, int* $\rho_3$  *pp2) {
2     *pp1 = *pp2;    // but  $\rho_1 \neq \rho_3$ 
3 }
4
5 void foo(int c) {
6     int* $\rho_H$  x = new 5;
7     'L: {
8         int* $\rho_L$  y = rnewL 8;
9         if (c == 1)
10             cp(&x, &y);
11     }
12     //read *x
13 }
```

Thus we should replace ρ_3 with ρ_1 for cp to type-check. Now, let's consider a safe version of the program:

```

1 void cp< $\rho_1$ >(int* $\rho_1$  * pp1, int* $\rho_1$  * pp2) {
2     *pp1 = *pp2;
3 }
4
5 void foo(int c) {
6     int* $\rho_H$  x = new 5;
7     'L: {
8         int* $\rho_L$  y = rnewL 8;
9         if (c == 1)
10             cp(&y, &x);    // safe, but errors!  $\rho_L \neq \rho_H$ 
11         //read *y
12     }
13     //read *x
14 }
```

The above program errors because the compiler doesn't "look" inside cp during the

call from `foo` due to performance reasons, nor does the signature carry enough information to guarantee that the body doesn't do anything unsafe (the assignment could be the other way round). So the only way of satisfying the compiler is to write it inline:

```

1 void foo(int c) {
2     int* $\rho_H$  x = new 5;
3     'L: {
4         int* $\rho_L$  y = rnew $_L$  8;
5         if (c == 1)
6             y = x;
7         //read *y
8     }
9     //read *x
10 }
```

The language (esp. the compiler) is a lot more complex than how it looks from the discussion up until now. Going into anymore details is beyond scope of this report. A few important areas that we skip are listed below:

- Unique pointers and many other types of pointers, which provide fine-grained control over memory while still guaranteeing safety.
- How soundness of a Cyclone program is analysed, using a concept called Linear Capabilities[7].
- Existential types and their interaction with regions. This was introduced in order to replace `void*` usage which is not type-safe.

The initial design did not provide thread-safety guarantees. But soon an extension was proposed[8] to address that. According to this new proposal, each region should maintain a list of threads accessing it at runtime. When a thread exits, it must remove its entry from those lists maintained by the regions it accesses. If a list becomes empty, free the region corresponding to it. Thus access to the list of threads must be protected by locks. These overheads were significant enough that it was never implemented.

In the next chapter, we will discuss an alternative to region based memory management, which avoids such book-keeping of allocated memory chunks at runtime; instead, the variables (or pointers) themselves are responsible for the resources they hold.

Chapter 5

Linear Type System

In Linear Type system (as described by Wadler[9]), every object must be used (a.k.a consumed) exactly once. Thus the system can safely reclaim an object’s resources after its use. This is a natural way of modeling the real world (and state machines). More concretely, an object, which represents a set of resources (a chunk of memory, an acquired lock, an open socket etc.), is *uniquely* bound to a variable. And the variable must be “used” exactly once. Next, we will look at how a purely linear typed language might behave using an approximate syntax of Rust, a language which has its roots in Linear Type system (although, Rust is more formally based on a concept called “External Uniqueness”[10]). Further, we will progressively relax the constraints of Linear Type system and approach the semantics of Rust. In the following example, assume that `add` and `print` are built-ins, and `print` accepts an arbitrary number of parameters:

```
1 | let a = 5;
2 | let b = 6;
3 | let c = add(a, b); // a and b are consumed
4 | print(c); // c is consumed
5 | //print(a, b); // error!
```

Since objects are uniquely bound to a variable, we may redefine the meaning of “consume” in terms of program variables: a variable is said to be consumed when it loses its binding. Let’s precisely define when that happens. Below are two rules of consumption:

1. A variable is consumed when it goes out of scope.¹

The underlying resources are released when this happens.

2. A variable is consumed when it appears in an expression.²

This means that, in all three cases shown below, the variable `a` gets consumed:

```
1 | f(a);
```

```
1 | let b = a;
```

```
1 | a;
```

This is the reason that a “consume” is called a “move” in Rust terminology.

¹This makes the “consume” implicit, and it might be appropriate to call it Affine Typing.

²While pattern matching, this may not hold true. Depending on the pattern, the object may be consumed, borrowed (next section), or left untouched (a no-op).

These rules (almost) enforces Linear Typing, and enables some neat analysis. In particular, spawning a thread involves a function call (and a closure), and multiple variables may get consumed in the process. Thus, objects can be passed between threads without any scope for data race, since there is no aliasing.

But, as you can see, this scheme would be too restrictive and impractical. Thus, Rust introduced `Copy` types which can *only* get consumed by going out of scope. Whenever a `Copy` typed variable is accessed, a duplicate copy is created and consumed. All numeric primitives are `Copy` types in Rust. It is important to note that a “copy” is not an alias of the original object; it is a separate independent object. For the sake of keeping examples simple, we will consider floating point numbers as non-`Copy` types.

Now let us see how a linear list would behave (assume that `get` is a built-in):

```
1 | let a = [1, 3, 5];  
2 | //let c = add(get(a, 0), get(a, 2)); // error: a is consumed twice
```

To get around this, we could redesign `get` to return both the original list as well as a copy of the element at the given index:

```
1 | let a0 = [1, 3, 5];  
2 | let (a1, b0) = get(a0, 0); // pattern-matched value-binding of a pair  
3 | let (a2, b1) = get(a1, 2);  
4 | let c = add(b0, b1);
```

But what if the elements are not copy-able (e.g. a list of file descriptors)? We could make `get` reject lists with non-`Copy` elements, and provide another built-in called `remove` which works as follows:

```
1 | let a0 = [1.0, 3.0, 5.0];  
2 | let (a1, b0) = remove(a0, 0); // a1 = [3.0, 5.0] and b0 = 1.0  
3 | let (a2, b1) = remove(a1, 1); // a2 = [3.0] and b1 = 5.0  
4 | let c = add(b0, b1);
```

This is still too restrictive, and most programs would need to be written in a “threaded-style” as shown above.

Borrowing

Wadler noted that “it is perfectly safe to have more than one reference to a value temporarily, as long as only one reference exists when the update is performed” [9]. This is the key idea behind the concept of Borrowing in Rust. Thus we will extend the rules of consumption to accommodate this:

3. A variable is not consumed in an expression when it is preceded by a borrow operator (discussed next).
4. A variable must not be consumed when a borrow is in effect.

`&` is a borrow operator. Below, `&a` denotes borrow of `a`. Technically, `b` is a pointer to the value `8.0`, but the responsibility of the underlying resource rests solely on `a`. And `b` is called a (shared) reference.

```
1 | let a = 8.0;  
2 | let b = &a;  
3 | let c = add(3.0, a); // error: a is borrowed
```

Points of note:

- `b` is a read-only view of `a`.
- There can be multiple shared references at the same time.
- Shared references are also `Copy` types. So `b` will not get consumed (and consequently the borrow does not end) until it goes out of scope.

In the next example, assume that `add_ref` takes a shared reference as the second argument and magically adds its value to the first, then returns the result.

```
1 | let a = 8.0;
2 | { // introduce a new scope
3 |     let b = &a;
4 |     let c = add_ref(3.0, b);
5 | } // borrow ends
6 | let d = add(3.0, a); // ok
```

Now, since it is safe to update an object when it is not aliased (i.e. no borrow is in effect), another kind of borrow was introduced: a “mutable borrow” which allowed in-place updates. `&mut` is the operator for mutable borrowing. In the next example, assume that `add_mut` takes in a mutable reference and updates it in-place.

```
1 | let mut a = 8.0; // mut marks ‘a’ as in-place updateable
2 | add_mut(&mut a, 4.0); // a mutable reference is created and consumed
3 | print(a); // prints 12.0
```

An object’s mutability is strictly unique. That is, an object cannot be mutably borrowed while another borrow (mutable or not) is in effect and vice versa. To see why they should be unique, consider the following example (assume that `square` brackets introduces a dynamic list; `truncate` resizes the list to a new length; and `print` will automatically read the value pointed to by a reference that is passed to it):

```
1 | let a = [1.0, 3.0, 5.0];
2 | let b = get_ref(&a, 2); // b = &a[2], indirectly borrowing ‘a’
3 | truncate(&mut a, 1); // drop the last two elements
4 | print(b); // possible dangling pointer dereference
```

The above program will not compile, of course. This means that Rust provides two guarantees:

- If you hold a shared reference (`&T`), you are guaranteed that it will not become invalid as long as you hold it.³
- If you hold a mutable reference (`&mut T`), you are guaranteed that you are having *exclusive access* to it. Thus, even destroying (or rewriting) a part of it will not affect any other references that are alive at that moment.

Since a mutable reference is non-`Copy`, ideally, the borrow should end the first time it is consumed and the “lender” (or “owner”) variable should become accessible. But this is not the case because keeping track of borrows through flow/liveness analysis is probably

³Rust does not guarantee that it will not be mutated because certain types may have “internal mutability”. Examples include wrappers which provide dynamic borrow-checking and mutual exclusion among others. But Rust guarantees that such mutations will not overlap and cause *Undefined Behavior*.

difficult to prove correctness of (but this is a planned feature). Anyway, as of now, this is where Rust meets Regions. Rust uses an algorithm based on Region Inference to keep track of borrows.

All borrows are associated with a lexical scope called the “lifetime of borrow”, and it represents the span of effect of that borrow. The reason for associating a lexical scope is so that lifetimes can have subtype (or outlives) relation between each other (no partial overlaps). This means that the following code is invalid:

```
1 | let mut a = 8.0;
2 | let b = &mut a;
3 | if condition {
4 |     add_mut(b, 2.0); // consume b4, but borrow does not end!
5 |     let c = &mut a; // error: a is borrowed
6 | }
```

The lifetime of a reference is upper-bounded by the scope of the owning variable. And inside the lifetime of the borrower, the object cannot be consumed (consumption rule 4).

```
1 | let b; // defer initialization5
2 | {
3 |     let a = 8.0;
4 |     b = &a; // error: scope of a is too small
5 | }
```

Reborrowing

Since mutable references cannot be `Copy`, we can’t do the following:

```
1 | fn add_six(b: &mut u32) {
2 |     add_mut(b, 4.0); // consume b
3 |     //add_mut(b, 2.0); // error: b was consumed
4 | }
```

Therefore, Rust provides a mechanism called “reborrowing”:

```
1 | fn add_six(b: &mut u32) {
2 |     add_mut(&mut *b, 4.0); // reborrow and consume the reference
3 |     add_mut(b, 2.0); // ok
4 | }
```

Reborrow is like dereference + borrow, but due to the 3rd rule, the dereference does not result in consume semantics (copy or “move”). And of course, reborrowing does *not* allow aliasing mutability:

```
1 | let mut a = 8.0;
2 | let b = &mut a;
3 | {
4 |     let c = &mut *b; // reborrow
5 |     //add_mut(b, 2.0); // error: *b is borrowed
6 | }
7 | add_mut(b, 2.0); // ok
```

⁴This is not exactly true since Rust automatically makes it a reborrow (discussed next). To keep things explicit, we will ignore that feature of Rust.

⁵`b` need not be marked as mutable, because through data-flow analysis, Rust can verify that `b` is not accessed while uninitialized, and that it will be initialized exactly once.

Lifetime

In Rust, *concrete* lifetimes are hidden from the programmer except the so-called `'static` which indicates forever-lifetime. The programmer is supposed to play with abstract lifetimes, as parameters in a function or type. For example, the function signature of the function `add_six` (above) may be rewritten as follows with explicit lifetime annotation:

```
1 fn add_six<'a>(b: &'a mut u32) { ... }
```

The lifetime parameter `'a` is kept abstract, even during analysis of the above function, with the assumption that `'a` outlives the entire scope of the function. Do note that, similar to Cyclone, Rust does not do whole-program lifetime inference. All inference takes place at an intra-procedural level, and therefore the programmer is required to insert explicit lifetime annotations wherever a set of default annotation rules is not appropriate. Thus, at the site of a function call, the compiler need to only check for inconsistencies with the lifetime constraints given in the function signature. We will take a closer look at this, but first, let us see how the compiler assigns concrete lifetimes to a simple program:

```
1 let mut a = 8.0;
2 {
3     let b = &mut a;
4     add_mut(&mut *b, 6.0);
5     add_mut(b, 2.0);
6 }
```

This is viewed by the compiler as:

```
1 'x: {
2     let mut a = 8.0;
3     'y: {
4         let b = &'y mut a;
5         'z: {
6             add_mut(&'z mut *b, 6.0);
7         }
8         add_mut(b, 2.0);
9     }
10 }
```

Note that the short scope of `'z` indicate that the reborrow is short-lived, after which `b` can be used (i.e. `b` becomes live). It is worth emphasizing that scopes *cannot* be assigned concrete names by the programmer as shown above.

Now, let's try a slightly more complex program involving two functions:

```
1 fn max_ref<'a>(x: &'a u32, y: &'a u32) -> &'a u32 {
2     if *a > *b { // side note6
3         return a;
4     } else {
5         return b;
6     }
7 }
```

⁶A dereference (without reborrow) is valid only if it is either a reference to a `Copy` type, or if it is a mutable reference and is used as an *lvalue*.

```

1 fn main() {
2     let u = 8;
3     let ur = &u;
4     {
5         let v = 4;
6         let vr = &v;
7         let mr = max_ref(ur, vr);
8         print(mr);
9     }
10 }

```

The compiler will view the `main` function as follows:

```

1 'a: {
2     let u = 8;
3     'b: {
4         let ur = &'b u;
5         'c: {
6             let v = 4;
7             'd: {
8                 let vr = &'d v;
9                 'e: {
10                    let mr = max_ref(ur, vr);
11                    print(mr);
12                }
13            }
14        }
15    }
16 }

```

When the lifetime constraints (such as lifetime equality assertion) does not match that of the actual arguments (`ur` and `vr` having different associated lifetimes), the compiler tries to make them match, subject to a set of variance rules. In the above case, as per one of the variance rules, `&'a T` is covariant w.r.t both `'a` and `T`. It means that the compiler is free to shrink the lifetime of `ur` to match that of `vr` (since shorter lifetime is more general).

Thus the type of `mr` will be `&'d u32`. There is a catch: the reference returned by `max_ref` effectively borrows both the arguments, due to the lifetime-equality assertion between the input parameters and the output. Consequently, `max_ref(ur, vr)` cannot escape beyond the scope of `'c`, even though the actual reference returned is `&u`.

Finally, let us look at the Rust-equivalent of the Cyclone-examples we discussed.

```

1 fn cp<'a>(pp1: &mut &'a i32, pp2: &&'a i32) {
2     *pp1 = *pp2;
3 }
4
5 fn foo(c: i8) {
6     let i = 5;
7     let mut x = &i;
8     {
9         let j = 8;
10        let y = &j;
11        if c == 1 {
12            cp(&mut x, &y); // unsafe! doesn't compile!
13        }
14    }
15 }

```

```

14     }
15     //read *x
16 }

```

Here is its safe version:

```

1 fn cp<'a>(pp1: &mut &'a i32, pp2: &&'a i32) {
2     *pp1 = *pp2;
3 }
4
5 fn foo(c: i8) {
6     let i = 5;
7     let x = &i;
8     {
9         let j = 8;
10        let mut y = &j;
11        if c == 1 {
12            cp(&mut y, &x); // ok!
13        }
14        //read *y
15    }
16    //read *x
17 }

```

Points of note:

- In the function definition, the inner lifetimes of both references `pp1` and `pp2` are annotated to be the same. As we said before, the compiler tries to make the arguments match the constraints given in the function signature, and in the above case, by shrinking all bigger lifetimes to match the smallest one, subject to variance rules.
- As per those rules, `&T` is covariant w.r.t `T` but `&mut T` is invariant w.r.t `T`. Therefore, in the former example, the inner lifetime of `&mut x` is not shrunk to match that of `&y` (which would be unsafe), but in the latter example, the inner lifetime of `&x` is shrunk to match that of `&mut y`.

As you can see, these notions of linear typing (a.k.a ownership) and lifetimes only exist during compilation, and therefore runtime overheads are close to zero.

Rust also provides type abstraction and polymorphism via its “trait-based” type system. Traits are similar to Haskell type classes or Java interfaces. Rust’s type system (traits + ownership + lifetimes) is powerful enough to enforce thread-safety constraints without the compiler having special knowledge about threads. A discussion on traits is beyond the scope of this report.

Conclusion

In our short survey of literature on provably safe methods to manage memory, we found that seemingly unrelated concepts can be mixed together to create very powerful, efficient, and practical tools that provide strong safety guarantees as well as allow fine-grained resource management.

References

- [1] Mads Tofte and Jean-Pierre Talpin. *Region-Based Memory Management*. In Information and Computation 132(2), pages 109-176. 1997.
- [2] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. *A Retrospective on Region-Based Memory Management*. Higher-Order and Symbolic Computation (HOSC). 17(3): 245-265, September 2004, Copyright © 2004 Kluwer Academic Publishers.
- [3] Public source code repository: <https://github.com/melsman/mlkit>
- [4] Martin Elsman. *Program Modules, Separate Compilation, and Intermodule Optimization*. PhD thesis. Revised. Department of Computer Science, University of Copenhagen. January 1999.
- [5] Martin Elsman. *Garbage Collection Safety for Region-based Memory Management*. In Proceedings of ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'03). New Orleans, Louisiana, USA. January 2003.
- [6] Grossman, Dan, et al. *Region-based memory management in Cyclone*. ACM Sigplan Notices. Vol. 37. No. 5. ACM, 2002.
- [7] Fluet, Matthew, Greg Morrisett, and Amal Ahmed. *Linear regions are all you need*. In Programming Languages and Systems, pp. 7-21. Springer Berlin Heidelberg, 2006.
- [8] Grossman, Dan. *Type-Safe Multithreading in Cyclone*. ACM Workshop on Types in Language Design and Implementation, pages 13–25, New Orleans, LA, January 2003.
- [9] Wadler, Philip. *Linear types can change the world*. IFIP TC. Vol. 2. 1990.
- [10] Clarke, Dave, and Tobias Wrigstad. *External uniqueness is unique enough*. In ECOOP 2003–Object-Oriented Programming, pp. 176-200. Springer Berlin Heidelberg, 2003.